SIG-SWO-042-05

RDF グラフに対するキーワード検索の高速化と省メモリ化

Fast Memory-Saving Keyword Search for RDF Graphs

浜松 良樹 [1*]　兼岩 憲 [1]
Yoshiki Hamamatsu[1]　Ken Kaneiwa[1]

[1] 電気通信大学大学院情報理工学研究科情報・ネットワーク工学専攻
[1] Department of Computer and Network Engineering, Graduate School of Informatics and Engineering, The University of Electro-Communications

Abstract: セマンティックウェブの普及と発展によって，今日では膨大な量の RDF データがウェブ上に存在する．RDF ストアでは，SPARQL クエリによる検索を前提にしており，膨大な RDF データに対する効率的な検索手段が求められる．SPARQL の利用にはデータセットの内容やクエリの文法に精通している必要がある一方で，キーワード検索は単語の入力だけで容易に検索できる．キーワード検索は，実装上 SPARQL クエリによって擬似的に実行できる．しかし，そのクエリの実行ではキーワードを検索するプロセスが非効率で時間がかかる．本研究では，インメモリ型 RDF ストア上で動作する高速なキーワード検索の方法を提案する．実験により，いくらかの RDF ストアで SPARQL クエリを用いるよりも高速かつ省メモリでキーワード検索が可能であることを示す．

1　はじめに

セマンティックウェブ [1] を実現するために，ウェブ上でリソース間の関係を示す RDF(Resorce Description Framework) [2, 3] データモデルがある．近年では，図書情報や地理情報などを始め，多くの機関で RDF によるリンクトデータが公開されている．例えば DBpedia[4] は Wikipedia から生成されたデータセットで，人物や地理など様々な事柄の関係が記述されている．そうした変化に伴って，大規模な RDF データを処理することができる RDF ストアの需要も高まっている．

RDF データの検索には，問い合わせ言語 SPARQL[5] を用いてクエリ検索する．しかし，クエリ発行には，SPARQL クエリの文法とデータセットのドメインに関する知識のそれぞれが必要となる．一方キーワード検索は，調べたい事柄に関するキーワードをいくつか入力するだけで検索できる．大規模 RDF データのためのキーワード検索の研究には，k-NK[6] 検索がある．k-NK 検索は専用のインデックスを作成することによって，高速なキーワード検索を実現するが，インデックスの作成に非常に時間がかかる．

FROST[7, 8, 9, 10] は，藤原らが開発した大規模 RDF グラフのための RDF ストアである．FROST は省メモリのインデックス構造によってインメモリで動作し，かつ SPARQL クエリを高速に解決できる．しかし，FROST はキーワード検索ができないので，SPARQL クエリで擬似的にキーワード検索することになるが，非効率で時間がかかる．同様に，Jena[11] や RDF4j[12] などの RDF ストアでも，SPARQL クエリによってキーワード検索を擬似的に実行できるが，クエリ解決の仕組み上，効率的だとは言い難い．

本研究では，RDF データを圧縮して省メモリで格納した RDF ストアにおいて，SPARQL クエリよる擬似的な方法よりも高速なキーワード検索を実現する．そのため，リレーインデックスにより RDF データを効率的に圧縮した RDF ストアの FROST を改良した K-FROST を提案する．K-FROST では，キーワード検索のための専用のインデックスは構築せず，検索時に距離付き到達可能リストを作成し検索する．この到達可能リストには，キーワードに対応するリソースから到達可能な目的語リソースを，最短距離ごとに記憶する．この距離情報を元にトリプルパターンへの代入を制限して，枝刈りを行う．

本稿の構成は，以下のとおりである．まず 2 章で RDF に関する基礎的な知識と，SPARQL クエリを用いた擬似的なキーワード検索の問題点を明らかにする．それらの問題点を踏まえて，3 章で FROST によって効率的なキーワード検索を可能にする技術を示す．4 章で，本手法とクエリによる擬似検索との比較を行い，最後に 5 章で結論を述べる．

*連絡先：
電気通信大学大学院情報理工学研究科情報・ネットワーク工学専攻
〒 182-8585 東京都調布市調布ヶ丘 1-5-1
E-mail: hamamatsu@sw.cei.uec.ac.jp

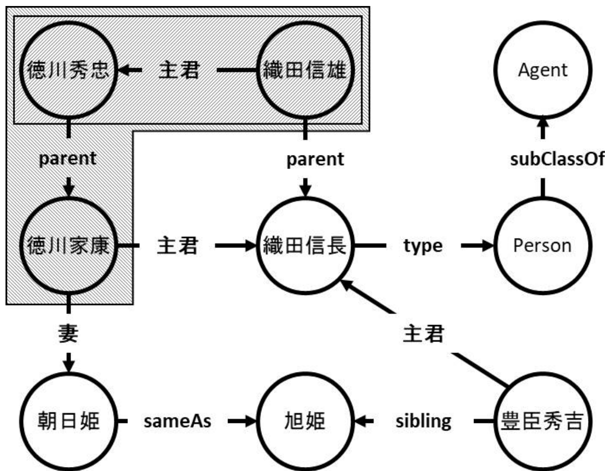


図 1: RDF グラフの例

2 RDF とキーワード検索

2.1 RDF グラフ

RDF はセマンティックウェブを構築する基盤技術の 1 つでありメタデータ，特にウェブ上でリソース間の関係を記述するために用いられる．RDF は主語 (subject)，述語 (predicate)，目的語 (object) の三つの要素からなるトリプルと呼ばれる形式で主語と目的語の関係を表現する．以降本論文では，主語を s，述語を p，目的語を o とし，RDF トリプルを (s p o) の形で表記する．s は URI(Uniform Resource Identifier)[13] で表現されたリソースの集合 R か，URI を持たず直接参照できない空白ノードの集合 B のどちらかの要素である．p は R の要素であり，o は R か B，リテラルの集合 L のいずれかの要素である．

RDF データ T は，RDF トリプル ($s\,p\,o$) の有限個の集合である．このとき s，p，o について，$s \in S(= R \cup B)$，$p \in P(= R)$，$o \in O(= R \cup B \cup L)$，$T \subseteq S \times P \times O$ を満たす．また RDF データ T について，s と o をグラフの頂点，p を辺とみなすことで，RDF グラフ G を構成する．図 1 は RDF グラフの一例である．

2.2 キーワード検索

キーワード検索とは，任意の複数個のキーワードを用いて情報を検索する手法である．RDF グラフ上の top-k キーワード検索は，RDF グラフからキーワードに対応する頂点を全て含むような部分グラフを k 個検索することと定義される [14]．本研究では，入力するキーワードの数を二つに制限し，キーワードに関連したリソース間のすべての最短経路を検索する．最短経路キーワード検索を次のように定義する．

定義 2.1 (最短経路キーワード検索) 任意の二つのキーワード q と w が与えられたとする．q,w それぞれについて，リソースを指す URI かリテラルを示す文字列とキーワードが部分一致するリソースを，キーワードに対応したリソース r_q，r_w と定め，その集合を R_q,R_w とする．このとき，$r_q \in R_q$ と $r_w \in R_w$ を結ぶ，長さ l 以下のすべての最短経路を検索する．

このように，キーワード検索は，(i) 入力キーワードの文字列を含むリソースを検索し，(ii) それらリソース間を結ぶ経路を探索する，二つの処理からなる．

例えば，図 1 の RDF グラフに対して，二つのキーワード「織田」と「徳川」で検索した場合を考える．経路検索の前にキーワードとリソースの対応を決定する．キーワード「織田」を含むリソースは「織田信雄」と「織田信長」の二つで，キーワード「徳川」を含むリソースは「徳川秀忠」と「徳川家康」の二つである．このとき検索される最短経路は，図 1 の四角で囲われた部分である．一つは，(織田信雄 主君 徳川秀忠) であり，もう一つは (織田信雄 主君 徳川秀忠)（徳川秀忠 parent 徳川家康）である．

2.3 三変数クエリによるキーワード検索

SPARQL は RDF データを検索するためのクエリ言語である．SELECT 文のクエリは，トリプルパターンの変数への代入を満たすリソースの組を出力する．トリプルパターンとは 0 個以上の変数が含まれたトリプルの連言である．このクエリは，SELECT 句と WHERE 句を核に構成される．SELECT 句では出力する変数を指定し，WHERE 句にトリプルパターンや FILTER による制約を記述する．

SPARQL でキーワード検索を実行するには，下記のようなクエリを発行する．このクエリは長さ 1 のトリプルパターンと，長さ 2 のトリプルパターンからなる．これらのクエリによって，二つのキーワードにそれぞれ対応したリソースを結ぶ，長さ 2 以下のパスが検索される．ただし，得られるパスが最短経路とは保証されない．以降この形のクエリを三変数クエリと呼ぶ．ここで FILTER 句では，経路が閉路を含まないように制限を与えている．

```
SELECT *
WHERE {
?qRes ?p ?wRes.
FILTER(REGEX(?qRes, q) &&
REGEX(?wRes, w) &&
?qRes != ?wRes)}
```

```
SELECT *
WHERE {
?qRes ?p1 ?o1. ?o1 ?p2 ?wRes.
FILTER(REGEX(?qRes, q) &&
REGEX(?wRes, w) &&
?qRes != ?o1 &&
?Res != ?wRes &&
?o1 != ?wRes)}
```

三変数クエリは，長さ l 以下のキーワードを含むリソース間の全経路（非最短経路を含む）を探索する．三変数クエリは RDF データの全件検索と同意であり，計算コストが非常に大きい．このため，キーワード検索に最低限必要な探索を効率的に行う方法が求められる．

3　キーワード検索の高速化

FROST は，メモリ効率の良いリレーインデックスを持つが,（SPARQL クエリによる擬似的な）キーワード検索は非常に遅い．本章では，FROST の省メモリ性を保ちながら，高速なキーワード検索を実現する方法を述べる．

3.1　キーワードを含むリソースの抽出

FROST では URI が記録された辞書を調べ，キーワードに対応したリソースをすべてリストアップする．FROST の URI 辞書は，Front Coding[15] により圧縮される．Front Coding は差分符号化の一種で，前のデータとの差分のみを記録し圧縮する方法である．図 2 では，リソース URI が圧縮前の URI で，Cmn が一つ前の URI との共通文字列の文字数，Len が非共通文字列の文字数，Diff が非共通文字列である．URI 辞書には n 個のリソースが格納されて，全リソース r は，一つ前のリソースとの共通文字列の文字数を Cmn(r)，非共通文字列文字列を Diff(r) で示す．0 番目のリソース r(0) について，Cmn(r(0)) $=0$ である．また Len(r) は r の文字数であり，Len(r) $=$ Cmn(r)$+$Len(Diff(r)) を満たす．

アルゴリズム 1 は，圧縮された URI 辞書 D からキーワードを含むリソースを効率的に抽出するスキップ法を示す．この方法は，圧縮された共通文字列にキーワードを含むか否かにより，文字列一致の比較回数を削減する．共通文字列にキーワードの途中まで含む場合は，その続きを含むか調べて文字列比較のコストを軽くする．

アルゴリズム 1 では，まず D から i 番目のリソース r(i) を取り出す．Cmn(r(i)) $=0$ のとき，Diff(r(i)) にキーワードを含むか調べる．文字列比較の結果，Diff(r(i)) の j から m 文字目まで一致するとき，(j, m) を Map に保存する．ただし，Map は j の降順でソートされている．このとき Len(k) $= m-j$ であれば，リソース r(i) はキーワード含む (3,4 行目)．

リソースURI	Cmn	Len(Diff)	Diff
織田政権	0	4	織田政権
織田信忠	2	2	——信忠
織田信長	3	1	———長
織田秀信	2	2	——秀信
織姫	1	1	—姫
織る織田信長	1	5	—る織田信長

スキップ法

マッチングをスキップ

図 2: キーワード「織田」を含むリソースの抽出

Cmn(r(i)) >0 のとき，一つ前ののリソース r(i $-$ 1) と共通文字列があるので Map から j の値が大きい順に (j, m) を取り出す．Cmn(r(i)) $>m$ かつ，Len(k) $=m-j$ なら，リソース r(i) は共通文字列にキーワードを含む (7,8 行目)．Cmn(r(i)) $=m$ かつ Len(k) $>m-j$ のときは，共通文字列の途中までキーワードを含むため，Diff(r) の先頭からキーワードの残り文字と一致するか調べる．Diff(r) がその残り文字から始まるときは，リソース r(i) の共通文字列と非共通文字列にまたがってキーワードを含む（9 行目から 11 行目）．キーワードの残り文字と不一致ならば，Diff(r(i)) にキーワードが含まれるかを調べ，Map に $(j+$Cmn(r(i))$, m+$Cmn(r(i))$)$ を追加する（13 行目）．

i 番目のリソース r(i) についての処理が終わったら，Map から Cmn(r(i)) $<j$ を満たすエントリをすべて削除し，i を更新する．アルゴリズムは $i>n$ を満たすと終了する (18 行目)．

例として，図 2 の辞書に対して,「織田」を含む URI を抽出する．1 行目で「織田政権」が 1-2 文字目にわたってマッチする．2 行目では共通文字列の「織田」の 2 文字が省略されている．既に 1 行目から 1-2 文字目にわたって「織田」が含まれるため，2 行目は非共通文字列の「信忠」の一致をせずにキーワードを含む．3, 4 行目も同様である．5 行目の「織姫」では 0 文字が「織」であることが 1 行目から分かるので，非共通文字列の 1 文字目が「田」，もしくは 2 文字目以降に「織田」が含まれればよい．1 文字目は「田」ではなく，またそれ以降にも「織田」が含まれないためキーワードを含まない．6 行目の「織る織田信長」も 5 行目と同様であるが，非共通文字列の 2 文字目以降に「織田」を含む．

Algorithm 1 SkipMethod

Require: URI 辞書 D, キーワード k
Ensure: キーワード k を含むリソースの集合 R_k

```
 1: Map の初期化
 2: for all i = 0 to |D| do
 3:   if Cmn(r(i)) = 0 then
 4:     Diff(r(i)) のマッチングと Map の更新
 5:   else
 6:     for all (j, m) ∈ Map do
 7:       if Cmn(r(i)) ≥ m かつ Len(k) = m − j then
 8:         R_k に r(i) を追加
 9:       else if Cmn(r(i)) = m かつ Len(k) > m−j then
10:         if Diff(r(i)).startWith(k.subString(m-j)) then
11:           R_k に r(i) を追加
12:         else
13:           Diff(r(i)) の検査と Map の更新
14:         end if
15:       end if
16:     end for
17:   end if
18:   Map の正規化と i の更新
19: end for
```

3.2 キーワード間の経路探索

キーワード間の経路を探索するために，一時的な補助データを作成する．RDF グラフは辺が述語に対応するので，RDF グラフの隣接性には，述語も表現しなければならない．隣接行列では，$|P| \times |O|$ 次正方行列が必要である．一方，隣接リストでは，(p, o) を記憶するだけであり，加えて RDF グラフは一般的に疎であるため，隣接行列よりもメモリ効率が良い．そこで，RDF グラフのために拡張した，述語付き隣接リストを次に定義する．

定義 3.1 (述語付き隣接リスト B^d) キーワード q を含む任意のリソース r_q について，B^d は r_q を始点とする長さ d の RDF パスの集合で，$B^d = \{(r_q, p_0, o_0, \ldots, p_{d-1}, o_{d-1}), \ldots\}$ である．ただし，$B^0 = \{r_q\}$ である．この述語付き隣接リストには長さ d の RDF パスが記憶されており，それは長さ $d+1$ のパスの部分パスとなる可能性がある．

アルゴリズム 2 は述語付き隣接リストを利用した経路検索である．キーワード q を含むリソース r_q を始点として，幅優先探索で r_w までのパスを探す．初めに $r_q \in R_q$ を長さ 0 のパスとし，B^0 に加える（2 行

Algorithm 2 Simple Keyword Search

Require: 長さ d, キーワード q, キーワード w
Ensure: パスの集合 T

```
 1: for all r_q ∈ R_q do
 2:   始点 r_q からなるパスを B^0 に追加
 3:   for all i = 1 → d  do
 4:     for all prefix ∈ B^{i−1} do
 5:       prefix を i+1 の長さに拡張し，path とする
 6:       if path の最後のリソースが r_q then
 7:         T に path を追加
 8:       end if
 9:       B^i に path を追加
10:     end for
11:   end for
12: end for
```

目）．次に，B^{i-1} からパスを一つ選択し *prefix* とする．インデックスを用いて *prefix* を長さ $i+1$ のパスへ拡張し *path* とする（5 行目）．*path* の最後のリソースが $r_w \in R_w$ であれば答えであるため，T に追加する (7 行目)．その後，次の拡張のために *path* を B^i へ追加する（9 行目）．

次の定理 3.1 のように，述語付き隣接リストの大きさは $O(n^d)$ である．キーワード検索では，経路の組み合わせ爆発によって保持しなければならないパスの数が膨大となる．

定理 3.1 (述語付き隣接リストの領域計算量) RDF データ T のトリプル数を n，キーワード検索の長さを d とする．このとき，述語付き隣接リスト B^d の領域計算量は $O(n^d)$ である．

そこで，述語付き隣接リストを簡略化し，組み合わせ爆発が起こらない距離付き到達可能リストを考える．この到達可能リストには，始点リソースから述語を保持しないで，到達可能な目的語リソースのみを，最短距離ごとに記憶する．

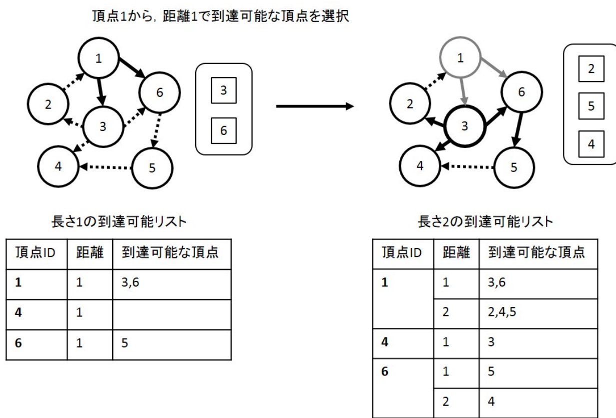
定義 3.2 (距離付き到達可能リスト A^d) キーワード q を含む任意のリソース r_q について，A^d は r_q から距離 d 以内で到達できるリソースの集まりで，$A^d = (A[0], A[1], \ldots, A[d])$ である．任意の距離 i について，$A[i]$ はリソース r_q から最短距離 i 丁度で到達可能なリソースの集合である．ただし $i = 0$ のとき，$A[0]$ は r_q のみで，$A[0] = \{r_q\}$ である．距離付き到達可能リストは最短距離のリソースを記憶できるため，同じリソースは一度しか出現しない．すなわち，$r \in A[i]$ かつ $r \in A[i']$ ならば必ず $i = i'$ を満たす．

アルゴリズム 3 は，距離付き到達可能リストを作成する．$A[d-1]$ からリソースを取得し，それを s とし

Algorithm 3 calcAList

Require: 距離 d,
　到達可能リスト $A^{d-1} = (A[0], A[1], \ldots, A[d-1])$
Ensure: 距離 d で到達できるリソースの集合 $A[d]$

1: **for all** $s \in A[d-1]$ **do**
2: 　$O_s \leftarrow s$ に隣接するリソースの集合
3: 　**for all** $o \in O_s$ **do**
4: 　　**if** o が $A[0], \ldots, A[d-1]$ のいずれにも含まれない **then**
5: 　　　o を $A[d]$ に追加
6: 　　**end if**
7: 　**end for**
8: **end for**



頂点ID	距離	到達可能な頂点
1	1	3,6
4	1	
6	1	5

頂点ID	距離	到達可能な頂点
1	1	3,6
	2	2,4,5
4	1	3
6	1	5
	2	4

図 3: 距離付き到達可能リストとその計算

てループさせる（1 行目）．SPO インデックスを用いて s に隣接するリソースを調べその集合を O_s とする（2 行目）．$o \in O_s$ について，o が $A[0], \ldots, A[d]$ のいずれの集合の要素でもなければ，$A[d]$ に加える (5 行目)．尚，SPO インデックスは FROST で RDF グラフを格納したものであり，$s \rightarrow p \rightarrow o$ の順番でリソースを辿ることで各トリプルが得られる．

図 3 は距離付き到達可能リストを作成する例である．あるキーワードが与えられ，ID 1，4，6 の頂点がキーワードを含むとする．頂点 ID1 について，そこから距離 2 で到達できる頂点を求める．長さ 1 の到達可能リストから，頂点 1 からは頂点 3，6 が繋がっていることが分かる．SPO インデックスを参照し，頂点 3 から新たに頂点 2 と頂点 4 が到達でき，頂点 6 からは新たに頂点 5 へ到達できる．結果として，頂点 1 から距離 2 で到達可能な頂点は，頂点 2，5，6 の 3 つであり，それを長さ 2 の到達可能リストに追加する．

以下の定理 3.2 は，述語付き隣接リストと比べて，距離付き到達可能リストの大きさが非常に小さいことを意味する．すなわち，A^d の大きさは $O(n)$ に抑えられ

距離付き到達可能リストA^l

$A[0]$	$A[1]$	…	$A[l]$
r_q	$r_0, \cdots r_{\vert A[1]\vert -1}$	…	$r'_0, \cdots r'_{\vert A[l]\vert -1}$

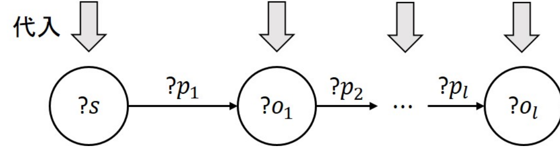


図 4: 解の生成

長さ d によって指数関数的に増大しない．

定理 3.2 (距離付き到達可能リストの領域計算量) RDF データ T のトリプル数を n とする．このとき距離付き到達可能リスト A^d の領域計算量は $O(n)$ である．

3.3 解の生成

距離付き到達可能リストを利用して, キーワード検索の解を生成する．長さ l のトリプルパターン t_l と到達可能リスト A^l が与えられたとする．t_l は 1 個の主語 s と l 個からなる p の集合 P_{t_l} と l 個からなる o の集合 O_{t_l} から構成される．このとき図 4 のように，距離付き到達可能リストと変数を対応させてリソースを代入する．s にはキーワード q に対応するリソース $r_q \in A[0]$ が代入され，$o_i (1 \leq i \leq l)$ には $A[i]$ に含まれるリソースが代入される．ただしパスの最後尾 o_l に代入されるリソースは $A[l]$ に含まれ，かつキーワード w を含む必要がある．このようにリソースを代入し，それが RDF データに存在すれば長さ l の答えとして出力する．すなわち，o_i について以下を満たす．

$$o_i \in \begin{cases} A[i] & (i \neq l) \\ A[l] \cap R_w & (i = l) \end{cases}$$

アルゴリズム 4 は，リソース r_q，r_w に対してキーワード検索の解の生成するアルゴリズムである．長さ $d \geq 1$ の解の生成は，長さ $d-1$ のパスに $A[d]$ のリソースを再帰的に付け加えていくことで生成する（1 行目から 8 行目）．$d = 0$ のときは，始点 r_q のみからなる特別なパターンとして定義される（13 行目）．

4 実験

本章では，K-FROST と三変数クエリによる SPARQL のキーワード検索との比較実験を示す．RDF ストアには，FRSOT 1.0.0，Jena 3.1.1 と RDF4j 2.1.4 を利用

Algorithm 4 makePath

Require: 長さ d, リソース r_q,r_w,
到達可能リスト $A^d = (A[0], A[1], \ldots, A[d])$
Ensure: r_q と r_w を結ぶ長さ d のトリプルパターンの集合 T

```
1: if d = l then
2:   for all prefix ∈ makePath(d − 1) do
3:     prefix の末尾に r_w が接続可能なら, 接続して T に追加
4:   end for
5: else if 1 ≤ d ≤ l then
6:   for all prefix ∈ makePath(d − 1) do
7:     s ← prefix.s
8:     for all s に隣接するすべてのリソース o について do
9:       if o ∈ A[d] then
10:        prefix の末尾に o を付け加え，t とする
11:        T に t を追加
12:      end if
13:    end for
14:  end for
15: else if d = 0 then
16:   始点 r_q のみからなる t を生成し T に追加
17: end if
```

する．実験には OS が Windows 8.1 Enterprise 64-bit (6.3, Build 9600)，CPU が Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (8 CPUs)，搭載メモリが 16GB である計算機を使用した．

使用したデータセットは DBpediaJapanese の 2014 年度版の一部で，2.55GB，620 万個のリソースからなる．データセット中には「織田」を含むリソースと「徳川」を含むリソースがそれぞれ 2,421 個，4,338 個存在する．このデータセットに対し，キーワード q を「織田」，キーワード w を「徳川」，長さを $2 \leq l \leq 5$ として検索した．その結果を表 1 に示す．なお，実験は 48 時間で timeout として中断する．

K-FROST では，長さ 2 からそれぞれ，521 個，3,028 個，13,663 個，45,698 個の最短経路が得られた．三変数クエリでは，長さ 2 からそれぞれ，542 個，5,585 個，66,368 個，782,302 個の経路を得られた．ただし K-FROST では最短経路のみを，三変数クエリでは最短経路以外も検索しているため経路の数が異なる．三変数クエリでは RDF4j の 52.93 秒が最速だが，K-FROST では 5.03 秒である．K-FROST では長さ 5 でも検索時間が 40 秒程度であったのに対し，他は約 1000 秒以上の時間がかかっている．FROST は FILTER 処理が低速であるため，他の二つの RDF ストアと比較して最も劣る．

表 1: 解決時間の比較

手法	解決時間 (秒)			
	長さ 2	長さ 3	長さ 4	長さ 5
K-FROST	**5.03**	**5.76**	**9.71**	**43.87**
FROST	673.84	7,155.77	78,323.18	timeout
Jena	129.99	190.75	278.24	994.44
RDF4j	52.93	262.08	1,789.15	16,10140

表 2: メモリ使用量の比較

長さ	メモリ使用量 (MB)				
	K-FROST		FROST	Jena	RDF4j
	全体	到達可能リスト			
1	**923.21**	0.01	**923.20**	4414.97	3078.11
2	**923.27**	0.06			
3	**923.47**	0.20			
4	**924.07**	0.60			
5	**925.79**	1.72			

表 2 は、メモリ使用量の比較である．K-FROST は FROST を利用しているため，データ読み込み直後のメモリ使用量は FROST と等しい．K-FROST と FROST は既存の二つの RDF ストアよりも省メモリである．K-FROST は，距離付き到達可能リストを逐次作成するので，各長さにおける生成後のメモリ使用量も調査した．到達可能リストを作成しても 1MB 程の大きさでしかなく，システム全体においても約 920MB であり，長さ 1 から 5 において，既存の RDF ストアよりも 3 分の 1 以下の少ないメモリで検索ができる．

5 結論

本研究では，スキップ法による URI 辞書からのリソース抽出と，距離付き到達可能リストを用いて，キーワード検索の高速化を提案した．SPARQL クエリでは，同じ経路を何度も調べ，最短経路探索に不向きな問題があった．距離付き到達可能リストにより，距離情報による探索の枝刈りを導入することで，不適切な解を除外し，パフォーマンスを向上させた．実験結果が示すように，K-FROST は，RDF データを圧縮して格納しながら解決時間の低下をもたらしている．

今後の課題としては，キーワード検索の拡張が挙げられる．現状では二つのキーワードしか対応しておらず，RDF グラフを逆方向に辿れないため，有益な解（部分

グラフなど）を見落としている可能性がある．この拡張は，さらに多くのメモリが必要になるので，距離付き到達可能リストの圧縮などが必要と考えられる．

参考文献

[1] 兼岩 憲：セマンティック Web とリンクトデータ, コロナ社 (2017)

[2] P. Haye: RDF Semantics, Technical report, W3C Recommendation (2004), http://www.w3.org/TR/2004/REC-rdf-mt-20040210

[3] 兼岩 憲：RDF と RDF スキーマの推論, 人工知能学会論文誌, Vol. 26, No. 5, pp. 473–481 (2011)

[4] DBpedia Community : DBPedia Japanese, http://ja.dbpedia.org/

[5] E. Prud'hommeaux, and A. Seaborne : SPARQL Query Language for RDF, Technical report, W3C Recommendation (2008), http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

[6] X. Lian , E. De Hoyos , A. Chebotko , B. Fu , and C. Reilly :
k-nearest keyword search in RDF graphs, Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 22, pp. 40–56 (2013)

[7] 藤原 浩司, 兼岩 憲：大規模 RDF グラフのための効率的なクエリ解決, 人工知能学会論文誌, Vol. 29, No. 4, pp. 364-374 (2014)

[8] 藤原 浩司, 兼岩 憲：大規模 RDF グラフに対する高速検索とデータ圧縮の両立, セマンティック Web とオントロジー研究会, SIG-SWO-A1402-08 (2014)

[9] 香川 俊幸, 兼岩 憲：FORST におけるデータストアの圧縮と読み込み手法, セマンティック Web とオントロジー研究会, SIG-SWO-040-06 (2016)

[10] FROST：SPARQL 検索エンジン , http://www.sw.cei.uec.ac.jp/frost/index-j.html

[11] Apache Jena : Jena A Semantic Web Framework for Java, http://jena.sourceforge.net

[12] The Eclipse RDF4J framework, http://rdf4j.org/

[13] T. Berners-Lee , R. Fielding , and L. Masinter : Uniform Resource Identifier (URI): Generic Syntax, Technical report, Network Working Group (2005), http://tools.ietf.org/html/rfc3986

[14] T. Tran , H. Wang , S. Rudolpha , and P. Cimiano :
Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data, IEEE International Conference on Data Engineering, Vol. 25, pp. 405–416 (2009)

[15] I. H. Witten, A. Moffat, T. C. Bell : Managing Gigabytes. Second edition (1999)